Editorial Analizando Muestras

José Santiago Vales Mena

Junio 16, 2024

Subtarea 3 [12 puntos]

En esta subtarea, los límites indican que hay una única máquina capaz de analizar todas las muestras. Por lo tanto, hay que rotar las muestras hasta que todas hayan pasado por esa máquina. La mejor forma de hacerlo es haciendo $N-1$ movimientos ya sea a la izquierda o a la derecha.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int N;
    cin >> N;

    cout << N - 1 << '\n';
    return 0;
}
```

Subtarea 1 [7 puntos]

Cuando $N = 2$, basta con verificar si es necesario rotar los elementos una vez o no. Para ello, simplemente hay que comprobar si el elemento i se encuentra dentro de la i-ésima lista.

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, sz, en_lista, x;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N;

    en_lista = 0;
    for (int i = 0; i < N; i++) {
        cin >> sz;
        while (sz--) {
            cin >> x;
            if (x == i) en_lista++;
        }
    }

    if (en_lista == N) cout << 0;
    else cout << 1;
```

```
    return 0;
}
```

Subtarea 2 [20 puntos]

Para esta subtarea, basta darse cuenta de que nunca conviene rotar en sentido contrario. Es decir, tus posibles respuestas son:

- No rotar.
- Rotar una vez a un lado.
- Rotar dos veces a un lado.

Entonces, podemos hacer un análisis como el anterior para ver cuántos están en su lugar, cuántos están fuera por 1 rotación a la izquierda y cuántos por una rotación a la derecha.

```
#include <bits/stdc++.h>
using namespace std;

int N, sz, no_rot_count, rot_iz_count, rot_der_count, x;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N;

    for (int i = 0; i < N; i++) {
        cin >> sz;
        bool no_rot = false;
        bool rot_iz = false;
        bool rot_der = false;
        while (sz--) {
            cin >> x;
            if (x == i) no_rot = true;
            if (x != (i + 2) % N) rot_der = true;
            if (x != (i - 2 + N) % N) rot_iz = true;
        }
        if (no_rot) no_rot_count++;
        if (rot_iz) rot_iz_count++;
        if (rot_der) rot_der_count++;
    }

    if (no_rot_count == N) cout << 0;
    else if (rot_der_count == N || rot_iz_count == N) cout << 1;
    else cout << 2;

    return 0;
}
```

Subtarea 4 [15 puntos]

En esta subtarea, siempre conviene rotar a la derecha. Las restricciones aseguran que rotando 100 veces a la derecha, todas las muestras quedarán analizadas. Como $N \geq 10^4$, se necesitan al menos $10^4 - 100$ movimientos a la izquierda para analizar una muestra que no esté ya en su lugar.

Para cada uno de los elementos, obtenemos la mínima cantidad de movimientos que debemos hacer a la derecha para que quede analizado, y finalmente sacamos el máximo de ellos. Esto solo requiere iterar por los números, lo que tiene una complejidad de $O(M)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, sz, x;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N;

    int ans = 0;
    for (int i = 0; i < N; i++) {
        int mini = N;
        cin >> sz;
        while (sz--) {
            cin >> x;
            mini = min(mini, x - i);
        }
        ans = max(ans, mini);
    }

    cout << ans;
    return 0;
}
```

Subtarea 5 [40 puntos]

Una clave para solucionar este problema es darse cuenta de que la solución óptima se encuentra haciendo L giros a la izquierda y R giros a la derecha desde la configuración inicial. Es decir, si hacemos una cantidad X de movimientos a la izquierda y luego un giro a la derecha, nunca será bueno volver a hacer un giro a la izquierda después de ese giro a la derecha.

Entonces, el problema se reduce a calcular hasta dónde debemos llegar a la izquierda (L) y hasta dónde queremos llegar a la derecha (R). Una vez que conocemos esos datos, podemos deducir que nuestra mínima cantidad de movimientos se ve como $2\min(L, R) + \max(L, R)$ (rotar hacia el más cercano y luego en la dirección contraria hasta llegar al siguiente).

Podemos aplicar un algoritmo de fuerza bruta que revise todas las posibilidades de las parejas (L, R) que pueden ser $0 \leq L, R < N$. Un total de N^2 parejas.

Para guardar los datos, no podemos crear una matriz de $N \times M$, pero sí podemos tener algo llamado una *lista de adyacencia*, que es un arreglo de vectores donde cada vector es del tamaño justo y necesario para ocupar los números que nos darán. Esto hace que nuestra memoria sea $O(M)$ en lugar de $O(N^2)$.

Finalmente, hacemos una función `revisa(L, R)` que nos determina si todas las muestras son analizadas si hacemos esa cantidad de rotaciones. Solo itera por todos los números y verifica si rotando L o R veces se llega hasta allí. El chequeo tiene una complejidad de $O(M)$ (donde $M = l_0 + ... + l_n$), y hacerlo para cada pareja nos da una complejidad total de $O(N^2 M)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, sz, x;
vector<vector<int>> vec;

bool revisa(int L, int R) {
    int analizados = 0;

    for (int i = 0; i < N; i++) {
        sz = vec[i].size();
```

```
        bool analizado = false;
        for (int j = 0; j < sz; j++) {
            int movL = (i - vec[i][j] + N) % N;
            int movR = (vec[i][j] - i + N) % N;
            if (movL <= L || movR <= R) analizado = true;
            if (analizado) break;
        }
        if (analizado) analizados++;
    }
    return analizados == N;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N;

    vec.resize(N);
    for (int i = 0; i < N; i++) {
        cin >> sz;
        vec[i].resize(sz);
        for (int j = 0; j < sz; j++) {
            cin >> vec[i][j];
        }
    }

    int ans = N;
    for (int L = 0; L < N; L++) {
        for (int R = 0; R < N; R++) {
            if (revisa(L, R))
                ans = min(ans, 2 * min(L, R) + max(L, R));
        }
    }

    cout << ans;

    return 0;
}
```

Subtarea 6 [100 puntos]

Solución 1

La primera solución se basa en reducir la complejidad de la subtarea 5. Algo que nos gustaría poder hacer es utilizar la información de (L, R) para responder $(L, R+1)$. Es decir, si vemos que con (L, R) no se analizaron todas las muestras, ¿cuántas nuevas se analizan con la rotación $R+1$? Con la estructura que tenemos no es tan fácil responder a esa pregunta. Entonces, haremos un cambio de estructura, y tendremos N listas que representen las N rotaciones a la izquierda y otras para la derecha. Por ejemplo, digamos que tenemos el siguiente caso:

- $0 : 3, 6$
- $1 : 2, 5$
- $2 : 0, 4$

con $N = 7$. Entonces el elemento 0 se analiza con 2 o 5 rotaciones a la derecha, el elemento 1 con 1 o 4, y el elemento 2 con 2 o 5. Entonces nuestra lista de rotaciones a la derecha se vería algo como:

- $0 : \{\}$
- $1 : \{1\}$

- $2 : \{0, 2\}$
- $3 : \{1, 2\}$
- $4 : \{0\}$
- $5 : \{0, 1\}$
- $6 : \{2\}$

Con esa información, podemos contar fácilmente cuántos elementos se analizan con k rotaciones, y obtener el resultado rápidamente con una cantidad de tiempo lineal y no cuadrática, como hasta ahora.

```
#include <bits/stdc++.h>
using namespace std;

int N, sz, pos;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N;
    vector<vector<int>> adj(2*N + 1);

    for(int i = 0; i < N; i++) {
        cin >> sz;
        for (int j = 0; j < sz; j++) {
            cin >> pos;
            int r = (pos - i + N) % N;
            int l = (N - r) % N;
            // Aqui unimos las dos listas de la posicion
            // [0, N-1] son rotaciones de L y  de la
            // [N + 1, 2 * N - 1] son rotaciones de R.
            adj[N + r].push_back(i);
            if (l > 0)
                adj[N - l].push_back(i);
        }
    }
    vector<int> cubeta(N);
    int pointer2 = N, distintos = 0;

    // Contamos el hacer L rotaciones a la izquierda.
    for (int i = 0; i <= N; i++) {
        sz = adj[i].size();
        for (int j = 0; j < sz; j++) {
            int idx = adj[i][j];
            if (cubeta[idx]++ == 0)
                distintos++;
        }
    }

    int ans = N;
    for (int i = 1; i <= N; i++) {

        // Eliminamos la fila i-1
        sz = adj[i-1].size();
        for (int j = 0; j < sz; j++) {
            int idx = adj[i-1][j];
            if (--cubeta[idx] == 0)
                distintos--;
        }

        // Agregramos la fila pointer2 hasta tener todas analizadas
```

```
        while (distintos < N) {
            pointer2++;
            sz = adj[pointer2].size();
            for (int j = 0; j < sz; j++) {
                int idx = adj[pointer2][j];
                if (cubeta[idx]++ == 0)
                    distintos++;
            }
        }
        int act_ans = 2 * min(N - i, pointer2 - N) + max(N - i, pointer2 - N);
        ans = min(ans, act_ans);
    }

    cout << ans;

    return 0;
}
```

Claro, aquí está la mejora del texto para la Solución 2:

Solución 2

Existe una segunda forma de solucionar el problema. Cada una de las muestras tiene un número l_i y r_i asociados, que son la mínima cantidad de rotaciones a la izquierda y a la derecha que debo realizar para que la muestra sea analizada.

Ordenemos esas parejas de mayor a menor con respecto a l_i. Si mi solución fuera solo hacer rotaciones a la izquierda, basta con ver el l_i máximo (que es la primera posición). Ahora, imaginemos que no queremos hacer tantas rotaciones a la izquierda. Entonces, podemos tomar el segundo mayor l_i de rotaciones a la izquierda, pero ahora hay que hacer una cantidad R de rotaciones a la derecha, pero esa cantidad tiene que ser el r_i asociado al l_i máximo.

Por tanto, podemos ir recorriendo nuestra lista y decrementando la cantidad de rotaciones a la izquierda, y las rotaciones a la derecha quedarán determinadas como el máximo de rotaciones a la derecha de los que no tomamos.

```
#include <bits/stdc++.h>
using namespace std;

int N, sz, pos;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> N;
    vector<pair<int, int>> vec(N);

    for(int i = 0; i < N; i++) {
        cin >> sz;
        vec[i] = {N, N};
        while (sz--) {
            cin >> pos;
            vec[i].first  = min(vec[i].first,  (pos - i + N) % N);
            vec[i].second = min(vec[i].second, (i - pos + N) % N);
        }
    }

    sort(vec.begin(), vec.end(), greater<pair<int, int>>());

    int r = 0;
    int ans = vec[0].first;
    for (int l = 0; l < N-1; l++) {
```

```
        r = max(r, vec[l].second);
        ans = min(ans, 2 * min(vec[l+1].first, r) + max(vec[l+1].first, r));
    }
    r = max(r, vec[N-1].second);
    ans = min(ans, r);

    cout << ans;

    return 0;
}
```